Resource Efficient Real-time Operating System for Wireless Sensor Nodes

Xing Liu[#1], Kun Mean Hou[#2], Christophe de Vaulx[#3], Chengcheng Guo[*4], Hongling Shi[#5]

[#] *LIMOS Laboratory UMR 6158 CNRS, Blaise Pascal University*

Clermont-Ferrand, France

[1] liu@isima.fr

[2] kun-mean.hou@isima.fr

[3] devaulx@isima.fr

[5] shi@isima.fr

[*] *Laboratory of Internet and Information Technology, Wuhan University*

Wuhan, China

[4] netgcc@whu.edu.cn

Abstract—**The operating system (OS) for the wireless sensor nodes (WSN) can be classified into two types, the event-driven model and the thread-driven model. For the event-driven OS, the memory resource consumption is less since one shared run-time stack is used, but the real-time response cannot be supported as every task runs to completion before the next one is scheduled. For the thread-driven OS, the real-time reaction can be achieved by means of the thread pre-emption, however, the memory resource consumption is high as every thread needs to have its own run-time stack. Since the real-time reaction is needed by many WSN applications and most WSN platforms are constrained in the memory resources, it is essential to develop a real-time WSN OS with a low memory resources consumption, and in this paper, an OS named HEROS is developed to achieve this objective. For HEROS, it uses a hybrid scheduling policy: both the event-driven and thread-driven scheduling strategies are implemented. Moreover, a dynamic memory allocation mechanism is adopted to make the memory resources be utilized more efficiently. Furthermore, a resource-efficient middleware is developed to simplify the user application development process. The final evaluation results proves that HEROS is a real-time, resources efficient and easy-to-use OS which can support most WSN applications.**

I. INTRODUCTION

Wireless sensor network (WSN) has been used in widespread domains ranging from the precision agriculture to the military surveillance [1]-[4]. And the operating system (OS) is one of the key technologies for the software development on sensor nodes as an outstanding OS can not only manage the platform resources well but also provide a good service for the user application development.

There are several challenges for the WSN OS development. One is the constraint of the platform resources, because most sensor nodes are low cost ones with limited memory resources, e.g., the sensor nodes equipped with the AVR ATmega1281 microcontroller has only 8 kilobytes RAM. Thus, an excellent WSN OS should have a low memory resource consumption. The second challenge is the real-time reaction support, this is required by many WSN applications such as the engine control process in the industrial system, the heart pacemakers monitoring in the medical system, etc. So the hard real-time response is also expected for the WSN OS. The third challenge is related to the user application development process. Since the sensor node hardware and software platforms are diverse [5], it difficult for the users to develop the WSN applications as they are required to understand the low-level system details. Thus, how to enable an OS to provide an abstract and user-friendly development environment for the WSN users is also a considerable topic.

Currently, many WSN OSs have been developed [6]-[11], but they are either no hard real-time supported [6]-[8],[10] or high consumed in the memory resources [9,11], and are thus limited to be used for many WSN applications as well as many WSN platforms. Moreover, many OSs do not provide a user-friendly application development environment[6],[9,10]. Consequently, it is difficult for the users to develop as they are needed to understand the low-level system details for the application programming.

To address all these challenges above, a hybrid real-time operating system HEROS is implemented and presented in this paper. The key features of HEROS include the following aspects: Firstly, it uses a hybrid scheduling mechanism, both the event-driven and thread-driven mechanisms are implemented. Event-driven is the default scheduling model, whereas it can switch to the thread-driven model immediately when the real-time events are triggered. By this way, HEROS takes the advantages of both event-driven system's low memory resource consumption as well as thread-driven system's real-time reaction. Secondly, HEROS uses the dynamic memory allocation mechanism to make use of the platform resources more efficiently, with this mechanism, a memory space is allocated only when it is needed and will be released and reused by the others once it has become invalided. Thirdly, to provide a user-friendly development environment for the users, an efficient middleware as well as an application code GUI generation mechanism are implemented.

The structure of this paper is as follows: In section II, the related work is introduced. In section III, the hybrid scheduling mechanism of HEROS is presented. In section IV,

the dynamic memory allocation mechanism is discussed. In section V, the IPC mechanism is presented. In section VI, the method to provide a user-friendly development environment is investigated. Finally, it is the evaluation, the conclusion and ongoing work.

II. Related Work

For the current WSN OSs, they can mainly be classified into three types in terms of the scheduling model: the event-driven model, the thread-driven model and the hybrid model.

A. Event-driven OS

The event-driven model has been used in the OSs such as TinyOS [6], SOS [8], etc.. In event-driven OS, a set of event handlers are defined, each handler is related to an event and will be called after an event is triggered. Every handler runs to completion before the next one is invoked, the interruption is enabled during a handler's execution, but the pre-emption to the handler is not allowed. Since all handlers are executed one by one other than in parallel, only one global stack is needed in the event-driven OS. Thus, the memory consumption of event-driven OS is low. However, the hard real-time reaction cannot be supported as when a time-critical event occurs, the related event handler cannot be executed immediately, but should be deferred until the current executing handler runs to completion.

B. Thread-driven OS

Thread-driven model has been used in the OSs such as the MantisOS [9], uCOS[11], etc.. In the thread-driven OS, a serial of threads are defined, all the threads are executed concurrently by the thread switch, but not one by one as the event-driven OS. Thus, every thread should have a private run-time stack which is used to save the thread context when it is pre-empted and recover the execution context when resumes next time. Due to the thread pre-emption mechanism, the hard real-time response can be achieved in the thread-driven OS, however, more memory resources will be consumed if compared with the event-driven OS.

C. Hybrid OS

Hybrid scheduling OS implements both the event-driven and thread-driven models and are used rarely in some OSs such as the Contki [7] and the LIMOS [10]. In both these OSs, the thread-driven model is implemented upon the event-driven model, this means that when an event is triggered, its related handler can be processed by creating several child threads and run these threads concurrently. A particular advantage of the hybrid model in the Contiki and LIMOS is the OS flexibility, because the OS will become a thread-driven one if only one handler is defined in the system while it will become an event-driven one if each handler has only one thread. Thus, the system can be configured to work in different models in terms of the different application environments. However, the real-time response can still not be supported in these OSs as the low-level native system infrastructure is still the event-driven model.

III. System Scheduling Policy

The scheduling mechanism of HEROS is shown in Fig. 1. In HEROS, a set of events and event handlers are defined. Events are the system signals which indicate the conditions to take some system actions have been satisfied, it can be triggered by a key pressing, a packet reception from the wireless media, a expired system timer, etc. Handlers are a series of subroutines or methods. Every handler is bound to one event. Once an event is triggered, the handler matching to it will be invoked after this event is dispatched by the system scheduler.

In HEROS, the events are classified into three types: the common events, the soft real-time (SRT) events and the hard real-time (HRT) events. For the common event, after it is triggered, the related event handler is not required to be invoked immediately. In reverse, for the HRT event, once it is triggered, the relevant handler should be called as quickly as possible. For the SRT event, it needs to be processed within a given time, but a short time delay is allowed.

It is known that the event-driven scheduling model is low cost in the memory resources, but not real-time supported to the events. Whereas, the thread-driven scheduling model is real-time supported, but high memory resources consumed. Thus, in HEROS a hybrid scheduling model is designed and implemented with the objective of achieving a real-time system with a low memory resources consumption. Seen in the Fig. 1, for the common and SRT events, they are dispatched in HEROS by the event-driven model while for the HRT events, they are dispatched by the thread-driven model.

Since HRT events are rarely generated in the WSN applications, thus most of the time HEROS runs in event-driven scheduling model, and the memory resource consumption is low. Since the arriving speed of the keep coming common/SRT events can be more quick than the event handlers' execution speed, a common event queue and a SRT event queue are used to buffer the system events. After the events are generated, they will be posted into these queues and then extracted one by one by the event dispatcher. The priority of the SRT event queue is higher than the common event queue, thus, the events in the SRT queue are extracted in precedence. Once an event is extracted, it will be dispatched and the related event handler will be called.

Although HEROS runs in event-driven model in default, it can switch to the thread-driven immediately when it is necessary. Once a HRT event is triggered, the current executing common/SRT handler will suspend and its run-time context will be saved, and then the OS will switch to the thread-driven scheduling model. Under the thread-driven model, one thread will be created for one HRT event, and the function of this thread is to execute the handler related to this HRT event. Before a thread is created, an independent run-time stack will be allocated for it in a dynamic memory allocation way, and after a HRT handler runs to completion, the corresponding thread will be deleted and all this thread' memory resources will be released.

By the hybrid scheduling means, the hard real-time reaction can be achieved in HEROS, and most of the time, the OS runs

in event-driven model with less memory resource consumed. Although in the thread-driven model, more memory resources will be consumed as each thread should have its own private stack, it is still not a severe problem as HEROS switches to thread-driven model only when the HRT events occurs, moreover, the thread stack will be released quickly on the completion of the HRT event handler.

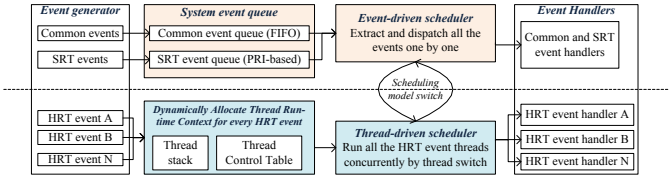

Figure 1. Hybrid Scheduling Model in HEROS.

IV. Dynamic Memory Allocation

The memory allocation mechanism is a key issue for the WSN OS as the RAM resources on the sensor nodes are precious, e.g., the node equipped with the microcontroller ATmega1281 has only 8KB RAM.

Currently, static memory allocation mechanism is used for most WSN OSs, such as the TinyOS [6], MantisOS [9], etc. For this method, the required memory resources are pre-reserved. This method is simple and the memory access process is efficient. However, two disadvantages exist. Firstly, the allocated memory area cannot be shared even if this area is not used frequently. Secondly, how much memory resource to be pre-reserved for a given system objective is difficult to decide, because in different applicable environment, the required memory resources can be different, e.g., in the WSN, when a node is configured to be an end device or to be a router, the size of the required memory resources for the packets buffering is different. One way to solve this problem is to pre-reserve the memory space as large as it's possible, whereas, the memory resources cannot be used efficiently in this case.

Except the static memory allocation, the dynamic block allocation is adopted in some OSs such as the Contiki [7] and the SOS [8].

In Contiki, only one kind of block is used and its size is equal to the largest memory size that the system objectives require. The drawback of this method is that a huge size memory block will be allocated even if the required memory space size is small, thus the efficiency of the memory resource utilization is low.

SOS improves the dynamic memory allocation method in Contiki by dividing the whole allocation space into three block sections, and in each section a different memory size block will be allocated, seen in Fig. 2. By this way, the block allocation will be done in different sections according to the required memory size. The improves the memory resource utilization efficiency, however, how much capacity to be reserved for each section is still difficult to decide as it depends on the practical application environments.

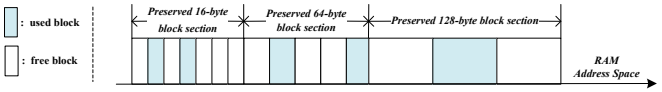

Figure 2. Example of an image with acceptable resolution

To avoid the drawbacks above, HEROS uses the dynamic block memory allocation mechanism as well, but it provides 4 kinds of memory blocks and there is no boundary limitation for the different kinds of blocks. Seen in Fig. 3, the 16-byte and 32-byte chunks are allocated from one direction in the heap while the 64-byte and the 128-byte chunks from another. The whole heap space is divided into three sections by both the virgin high and low address pointers, and the freed chunks are linked by the freed chunk queue. Every time a new chunk allocation is required, the freed chunk queue will be checked in precedence, if no proper chunk can be found, then the virgin area will be eaten into. And when a chunk is released, it will be inserted into the freed chunk queue.

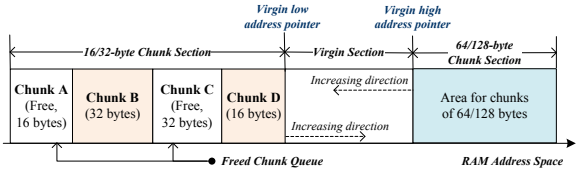

Figure 3. HEROS dynamic memory allocation in block.

V. Inter-handler Communication

Besides the scheduling policy and the memory management, the inter-process communication (IHC) is also important for an OS. In HEROS, IHC is a set of methods for the data exchange among multiple system handlers.

The key topic for the IHC is the receiver reception synchronization problem. This means that when the sender delivers a message, the receiver should be aware of receiving and processing it.

A common way to solve this synchronization problem is to provide an abstract layer between the sender and receiver to uncouple the sending and receiving process. For the sender, it sends the message into a dedicated message buffering space without considering whether the receiver is ready to receive or not. Then, for the receiver, it will check this message buffering centre in intervals and extract the message if it is observed. This method has been used in the OS such as the LIMOS [10], etc.

However, in HEROS, a positive notification method is used for the IHC implementation. When the sender needs to deliver a message to the receiver, it will create a IHC event, inside this event, the receiver's receipt handler, the message data address as well as the message length are recorded, seen in the Fig. 4. Then, this event will be posted into the system event queues and waited to be scheduled by the HEROS dispatcher. Once this event is dispatched, the receipt handler will be called to collect the sending message and process it. And in terms of the emergence of the sending message, the IHC event can be posted either into the common event queue or the SRT event queue.

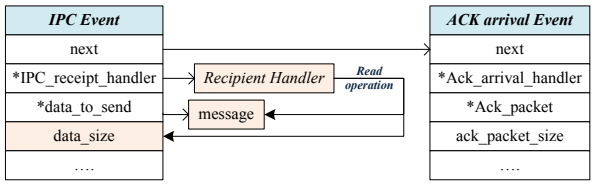

Figure 4. Example of an image with acceptable resolution

VI. USER-FRIENDLY APPLICATION DEVELOPMENT ENVIRONMENT

It is difficult for the users to develop on the sensor nodes as they are needed to understand the low-level hardware details as well as the OS features for the application programming [12]. In order to provide a user-friendly development environment for the WSN users, a HEROS middleware is implemented, this middleware separates the application part from the system part, and provides a set of system services from the system space to the application space. By this way, the application image is built independently from the system one and the users can program the applications without considering the low-level system details.

The elementary diagram of HEROS middleware is shown in Fig. 5. Four key topics should be considered for its design and implementation, they are the way to access the system functions from the application space, the way to pass the application parameters to the system space, the way to callback from the system to the application space as well as the way to support multitasking programming in the application space. And the work presented in this section is based on the AVR microcontroller with GNU GCC as the compiler.

A. Access to the system functions

As the application part is separated from the system part, how to pre-link the application function to a system function is the first topic. One way to solve this problem is to built the system space firstly to get an address list of the system functions, and then pre-link the application functions according to this list. However, in this case, once the system space is changed, the application functions need to be re-linked.

In order to improve the flexibility of the application code, in HEROS middleware, an intermediate function jump table is provided in the system space. With this table, the application function is pre-linked to a given address in this table, but not to the system functions directly. By this way, the modification to the system space will not cause the application code to become invalided in case that this jump table is put in a fixed system address.

B. Parameters passing from the application space to the system one

Once an application function is linked correctly to a given system function, the second problem to be solved is how to pass the function parameters from the application function to the system one.

After the code is built, it is observed from the AVR assembler language that in the application image the function parameters are put into the processor registers (begin from the "r24") before the function "CALL" directive is executed [13]. And after the "CALL" directive is loaded, the execution of the program code will jump to the corresponding system function by means of the intermediate function table. And at the beginning of the system function, it is found that the function parameters are also be extracted from the processor registers beginning at r24. This means that there is no parameters passing gap between the application and the system functions, and this reason results from the same compiler used in both the application and system spaces, thus the same operation rules are followed by both the two functions.

C. Callback from the system to the application space

With the above operations, the application space can access to the system space, but the reverse process is not enabled. However, the callback operation from the system to the application space is required in many cases, e.g., when an interruption occurs, the interruption routine service (ISR) will be invoked in the low-level system space. If the user wants to make program to process this interruption in the application space, then a callback operation should be done to redirect the interruption handling from the system ISR to the application ISR.

In HEROS middleware, the callback operation is achieved by the registration mechanism, seen in the Fig. 5 and 6. Any function in the application space can be registered into the system space by invoking the registration function, once it has been registered, its address will be acquired by the system functions, and then the callback operation from the system to the application space can be realized.

D. Multi-tasking applications support

For the WSN applications, it may have several independent application tasks. If the multi-tasking applications can be supported by the HEROS middleware, then only one application image other than several separated ones is needed.

To enable HREOS middleware support the multi-tasking applications, the registration mechanism similar to the callback registration method is used, but what is different is that the application handlers are registered into the system scheduling centre. And after the registration the system scheduler will be aware of this application handler and dispatch it at its turn.

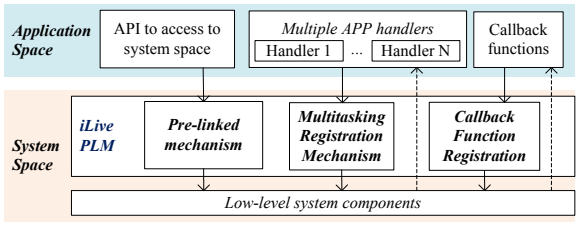

Figure 5. Elementary diagram of iLive middleware.

With the HEROS middleware implemented, an application program example can be as follows:

```
/* application program, built independently, and then uploaded. */
int main(void)
{
    /* register two application handlers. */
    multitasking_register(task_light_crtl_sensors, SSC_id);
    multitasking_register(task_packet_forwarding, PF_id);
    /* register an interruption handler to the low-level system. */
    callback_register(common_packet_arrival, CPA_id);
    /* configure and start the wsn. */
    wsn_start();
}
/* state machine programming for the application handlers. */
Int task_light_crtl_sensors(void)
{
    if(sensor_light_value > 100)   open_allSensors();
    else close_allSensor();
}
```

Figure 6. Application programming by means of HEROS middleware.

E. Performance Evaluation

In Contiki, the application space is also separated from the system space, but it uses the dynamic loadable mechanism (DLM) [14] other than the pre-linked mechanism (PLM).

In the DLM, the application image is built into a loadable Compacted Executable and Linkable Format (CELF) module. After it is loaded onto the target board, it will be linked firstly by the system dynamic linker before being executed. Thus, a dynamic linker should be implemented on the target device to resolve the function and variable references contained in the CELF module.

To evaluate the performance of HEROS PLM, it is compared with the Contiki DLM from the aspects of features, memory resource consumption and the application image size, with the comparison results shown below:

TABLE I
COMPARISON BETWEEN HEROS PLM AND CONTIKI DLM

Topics	Mechanisms	
	Contiki DLM	HEROS PLM
Application linking mechanism	Dynamic linking	Pre-linked
Callback from system to application space	Not supported	Supported
Multi-tasking programming in application	Not supported	Supported
Application image flexibility	Well	Normal
Required ROM (KB)	6.5	1.8
Required RAM (bytes)	27	16
Application image size	965	138

From the results, it can be seen that Contiki DLM does not consider the features such as the calling back operation from the system space to the application image, the multi-tasking programming in one application image, etc. As for the memory resource consumption, HEROS PLM consumes much less if compared with Contiki DLM, this is because the pre-linking process is done on the personal computer for the HEROS PLM while it is on the target sensor devices for the Contiki DLM. Moreover, to compare the application image size, the example in previous Fig. 6 is built by both the Contiki DLM and the HEROS PLM mechanisms, with the size result shown in the Table I. It can be seen that the PLM image size is smaller than the DLM one, this is because no function and variable references are needed to be contained in the PLM image. Certainly, for the application image's flexibility, the DLM image is better than the PLM one and this is the inherent advantage of dynamic loadable mechanism.

F. New code generation and updating

With the HEROS middleware, the application program becomes simple and the application image size is also small, thus it is feasible to update the WSN applications through the Internet by means of the webpage. And in order to further simplify the application programming process, a code generation graphic user interface (GUI) is developed to generate the application code by graph drawing. Seen in the Fig. 7, after the application program is generated through the GUI tools, it will be sent to the edge router by the Internet socket communication. On the edge router, this program will be built and pre-linked by HEROS PLM to produce the final application image. Later, this image will be updated onto the end-devices and executed. Currently, such a develop environment has been developing by us with the website address: edss.isima.fr (login id: liu@isima.fr, pwd: limos).

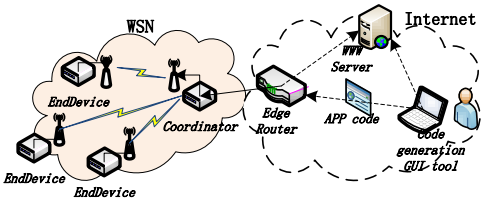

Figure 7. User-friendly development environment for iLive sensor nodes.

VII. EVALUATION

To evaluate the performance of HEROS, it is compared with Contiki and MantisOS from the aspects of OS features , the memory resource consumption and the portability.

A. Evaluation Platform

Currently, HEROS is implemented on the iLive node. ILive node is equipped with the microcontroller AVR ATmega1281, seen in the Fig. 8. It has 128 kilobytes of FLASH and 8 kilobytes of RAM. Besides, it is equipped with 11 sensors that include 1 temperature sensor, 1 light sensor, 1 air humidity sensor, 3 decagon sensors and 4 watermark sensors. And for the software development environment, the AVR-GCC is chosen as the compiler.

Figure 8. iLive sensor node

B. OS Features

The OS features comparison results are shown in the Table II. It can be seen that HEROS is a hybrid scheduling model OS, mantisOS is a pure thread-driven system. For Contiki, the native scheduling policy is event-driven, however, the thread-driven model is also implemented as a optional library [15]. For the memory allocation, Contiki also uses the dynamic block allocation, but only one type block is defined. For HEROS, four kinds of blocks are defined and the allocation is processed dynamically from two opposite directions, thus the memory resources are used more efficiently. As for the soft real-time response, it is achieved in HEROS by the high priority SRT event queue scheduling, in Contiki by the high priority polling mechanism, and in mantisOS by the thread switch. For the hard real-time response, it is supported only in HEROS and MantisOS due to the thread pre-emption operation. For the separation of the application space from the low-level system one, it is supported in HEROS and Contiki respectively by the HEROS middleware and Contiki DLM, but in MantisOS this separation is not supported and a monolithic system is used. For the WSN application reprogramming, the pre-linked application image is used in HEROS while the compacted ELF file is used in Contiki.

TABLE II
COMPARISON RESULTS OF OS FEATURES

Features	WSN OSs		
	HEROS	**Contiki**	**MantisOS**
Native scheduling policy	Hybrid, (event-driven + thread-driven)	event-driven	Thread-driven
Optional scheduling policy	No	Thread-driven	No
Memory allocation	Dynamic	Dynamic	Static
Soft real-time response support	Yes	Yes	Yes
Hard real-time response support	Yes	No	Yes
Uncouple application space from system space	Yes, by HEROS middleware	Yes, by Contiki	No, monolithic image
Application image for reprogramming	pre-linked image	compacted ELF image	diff-patch file

C. Memory Resource Consumption

The required ROM resource sizes of different OSs are shown in the Table III. It can be seen that HEROS consumes more ROM resources than MantisOS and Contiki, because both the event-driven and thread-driven scheduling models are implemented in it. For the RAM resources, MantisOS requires about 500 bytes for a common program to run, and it is mainly used for the thread control table, the thread stacks, etc. For Contiki, the required RAM size is *(26+4e+2p+s)*, where *e* represents the maximum size of the asynchronous event queue, *p* represents the maximum number of processes that the system is configured to have, and *s* represents the thread stack size. For HEROS, the kernel takes up 28 bytes, and then all the left objectives are allocated from the heap in dynamic allocation way. Since the memory resources in HEROS will be released once they are no more used, thus the memory resource consumption of HEROS is not high.

TABLE III
COMPARISON RESULTS OF KERNEL MEMORY RESOURCE CONSUMPTION

Features	WSN OSs		
	HEROS	**Contiki**	**MantisOS**
Required ROM (KB)	4.6	1.8	3.8
Required RAM (bytes)	$28 + S_h$	26 + 4e + 2p + s	500

D. Portability

The diversity of the hardware platforms in WSN makes the portability become important for the WSN OSs.

For HEROS, it is implemented as a general OS which can be used on diverse hardware platforms. And the platform-dependent code in HEROS involves: 1). the variable type, this is required because for different microcontroller, the length of the variable type can be different, thus the adaption to some data structures should be done, e.g., on the ARM AT91sam7s256 microcontroller, the integer type is 32-bits while on AVR ATmega1281 microcontroller it is 8-bits. 2). assembler language code inside the OS, e.g., the thread run-time context saving and recovery parts. 3). hardware-dependent part, such as the time interruption, system initialization ,etc. It is calculated that to port HEROS from the AVR ATmega1281 platform to the ARM AT91sam7s256 one, the lines of codes that needs to be adapted is about 60. Therefore, it is feasible and easy to use HEROS on different WSN platforms.

VIII. CONCLUSION AND ONGOING WORK

In this paper, we present a resource efficient real-time OS named HEROS. HEROS implements a hybrid scheduling model and uses the dynamic memory allocation mechanism. With these means, it can achieve the hard real-time reaction with a low memory resource consumption. Moreover, it provides a decent development environment for the users by implementing a middleware in the low-level system. The final evaluation results show that HEROS is a easy-to-use OS which can be used on different WSN platforms to support the different WSN applications.

The ongoing work mainly includes two aspects: on one hand, we will develop a fault-tolerant system by using HEROS on our multi-core sensor nodes. On the other hand, we will evaluate HEROS by means of the QEMU tool which is a generic and open source machine emulator [16].

ACKNOWLEDGMENT

We would like to thank the 'China Scholarship Council' (2009627016), the National Natural Science Foundation of China (NO. 60903195) and the 'Conseil Général du Puy-de-Dôme' (France) for their supports to this project.

REFERENCES

[1] K Shinghal, A Noor et al. "Intelligent Humidity Sensor For-Wireless Sensor Network Agricultural Application", Journal of Wireless & Mobile Networks, Vol. 3, No. 1, February 2011.

[2] A Flammini, Paolo Ferrari et al.. Wired and wireless sensor networks for industrial applications. In Microelectronics Journal, pp. 1322-1336, 2009.

[3] Norman A. Benjamin and Suresh Sankaranarayanan, "Performance of Wireless Body Sensor based Mesh Network for Health Application", Journal of Computer Information Systems and Industrial Management Applications, Vol.2, pp.020-028, 2010.

[4] I. Kirbas et al.. HealthFace: A web-based remote monitoring interface for medical healthcare systems based on a wireless body area sensor network. Journal EJEECS, pp. 629-638, 2012.

[5] Ana-Belén García-Hernando et al.. Problem Solving for Wireless Sensor Networks. Journal of Computer Communications and Networks. 2008.

[6] J. Hill et al.. System architecture directions for networked sensors. In Proc. ASPLOS, Cambridge, MA, USA, Nov. 2000, pp. 93–104, ACM.

[7] Dunkels et al.. Contiki - a lightweight and flexible operating system for tiny networked sensors. In Proc. ENC'04, Florida, USA, Nov. 2004.

[8] Chih-Chieh Han, et al., "A Dynamic Operating System for Sensor Nodes," In Proc. Int'l Conf. MobiSys'05, ACM Press, 2005, pp. 163-176.

[9] S. Bhatti et al. Mantis OS: An embedded multithreaded operating system for wireless micro sensor platforms. ACM/Kluwer Mobile Networks & Applications, 10(4):563–579, August 2005.

[10] Hai-Ying Zhou, Kun-Mean Hou, J.-P. Chanet, C. de Vaulx, G. De Sousa. LIMOS: A Tiny Real-Time Micro-Kernel for Wireless Objects. Wireless Communications, Networking and Mobile Computing, 2006. WiCOM'06. Wuhan, China

[11] uCOS, a real-time operating system. http://www.ucos-ii.com/.

[12] L. Bai, R. Dick, and P. Dinda. Archetype-based design: Sensor network programming for application experts, not just programming experts. In Proc. Int'l Conf. IPSN'09, San Francisco, CA, USA, April 2009.

[13] AVR user manual. http://www.nongnu.org/avr-libc/user-manual/pages.html.

[14] A. Dunkels et al.. Run-time dynamic linking for reprogramming wireless sensor networks. In Proc. SenSys, Boulder, Colorado, USA, 2006.

[15] A. Dunkels et al.. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In Proce. Intl' Conf. Embedded Networked Sensor Systems, SenSys 2006, USA, 2006.

[16] QEMU, a generic and open source machine emulator and virtualizer. http://wiki.qemu.org/Main_Page.